Chapter 9

Loops, Loops, Loops

In This Chapter

- Understanding loops
- Exploring the `for` loop
- Creating nested `for` loops
- Working a `while` loop
- Using a `do-while` loop
- Avoiding the endless loop

Programs love to do things over and over, mirthfully so. They never complain, they never tire. In fact, they'll repeat things forever unless you properly code instructions for when to stop. Indeed, the loop is a basic programming concept. Do it well. Do it well. Do it well.

A Little Déjà Vu

A *loop* is a section of code that repeats. How often? That depends on how you write the loop. Basically speaking, a loop involves three things:

- Initialization
- One or more statements that repeat
- An exit

The *initialization* sets up the loop, usually specifying a condition upon which the loop begins or is activated. For example, "Start the counter at 1."

The statements that repeat are contained in curly brackets. They continue to be executed, one after the other, until the exit condition is met.

The *exit condition* determines when the loop stops. Either it's a condition that's met, such as "Stop when the counter equals 10," or the loop can stop when a `break` statement is encountered. The program execution continues with the next statement after the loop's final curly bracket.

Having an exit condition is perhaps the most important part of a loop. Without it, the loop repeats forever in a condition called an *endless loop*. See the later section "Looping endlessly."

The C language features two looping keywords: `for` and `while`. Assisting the `while` keyword is the `do` keyword. The `goto` keyword can also be used for looping, though it's heavily shunned.

The Thrill of for Loops

A *loop* is simply a group of statements that repeats. You may want a set number of iterations, or the number of repeats can be based on a value. Either way, the `for` keyword helps set up that basic type of loop.

Doing something x number of times

It's entirely possible, and even a valid solution, to write source code that displays the same line of text ten times. You could copy and paste a `printf()` statement multiple times to do the job. Simple, but it's not a loop. (See Listing 9-1.)

Listing 9-1: Write That Down Ten Times!

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0; x<10; x=x+1)
    {
        puts("Sore shoulder surgery");
    }
    return(0);
}
```

Exercise 9-1: Create a new project using the source from Listing 9-1. Type everything carefully, especially Line 7. Build and run.

As output, the program coughs up the phrase *Sore shoulder surgery* ten times, in ten lines of text. The key, of course, is in Line 7, the `for` statement. That statement directs the program to repeat the statement(s) in curly brackets a total of ten times.

Exercise 9-2: Using the source code from Listing 9-1 again, replace the value `10` in Line 7 with the value `20`. Build and run.

Introducing the for loop

The `for` loop is usually the first type of loop you encounter when you learn to program. It looks complex, but that's because it's doing everything required of a loop — in a single statement:

```
for(initialization; exit_condition; repeat_each)
```

Here's how it works:

`initialization` is a C language statement that's evaluated at the start of the loop. Most often, it's where the variable that's used to count the loop's iterations is initialized.

`exit_condition` is the test upon which the loop stops. In a `for` loop, the statements continue to repeat as long as the exit condition is true. The expression used for the *exit_condition* is most often a comparison, similar to something you'd find in an `if` statement.

`repeat_each` is a statement that's executed once every iteration. It's normally an operation affecting the variable that's initialized in the first part of the `for` statement.

The `for` statement is followed by a group of statements held in curly brackets:

```
for(x=0; x<10; x=x+1)
{
    puts("Sore shoulder surgery");
}
```

You can omit the brackets when only one statement is specified:

```
for(x=0; x<10; x=x+1)
    puts("Sore shoulder surgery");
```

In this `for` statement, and from Listing 9-1, the first expression is initialization:

```
x=0
```

The value of the `int` variable *x* is set to `0`. In C programming, you start counting with 0, not with 1. You'll see the advantages of doing so as you work through this book.

The second expression sets the loop's exit condition:

```
x<10
```

As long as the value of variable *x* is less than 10, the loop repeats. Once that condition is false, the loop stops. The end effect is that the loop repeats ten times. That's because *x* starts at 0, not at 1.

Finally, here's the third expression:

```
x=x+1
```

Every time the loop spins, the value of variable *x* is increased by 1. The preceding statement reads, "Variable *x* equals the value of variable *x*, plus 1." Because C evaluates the right side of the equation first, nothing is goofed up. So if the value of x is 5, the code is evaluated as

```
x=5+1
```

The new value of *x* would be `6`.

All told, I read the expression this way:

```
for(x=0; x<10; x=x+1)
```

"For *x* starts at 0, while *x* is less than 10, increment *x*."

Listing 9-2 shows another example of a simple `for` loop. It displays values from `-5` through `5`.

Listing 9-2: Counting with a Loop

```
#include <stdio.h>

int main()
{
    int count;

    for(count=-5; count<6; count=count+1)
    {
        printf("%d\n",count);
    }
    return(0);
}
```

Exercise 9-3: Type the source code from Listing 9-2 into a new project. Build and run.

Exercise 9-4: Create a new project using the source code from Listing 9-2 as a starting point. Display the values from `11` through `19`. Separate each value by a tab character, `\t`. Use the `<=` sign for the comparison that ends the loop. Clean up the display by adding a final newline character when the loop is done.

- The `for` statement uses two semicolons to separate each item, not commas. Even so:
- It's possible to specify two conditions in a `for` statement by using commas. This setup is rather rare, so don't let it throw you. See the later section "Screwing up a loop" for an example.

Counting with the for statement

You'll use the `for` statement quite frequently in your coding travels. Listing 9-3 shows another counting variation.

Listing 9-3: Counting by Two

```
#include <stdio.h>

int main()
{
    int duo;

    for(duo=2;duo<=100;duo=duo+2)
    {
        printf("%d\t",duo);
    }
    putchar('\n');
    return(0);
}
```

Exercise 9-5: Create a new project using Listing 9-3 as your source code. Compile and run.

The program's output displays even values from `2` through `100`. The value `100` is displayed because the "while true" condition in the `for` statement uses `<=` (less than or equal to). The variable *duo* counts by two because of this expression:

```
duo=duo+2
```

In Line 9, the `printf()` function uses `\t` to display tabs (though the numbers may not line up perfectly on an 80-column display). Also, the `putchar()` function kicks in a newline character in Line 11.

Exercise 9-6: Modify the source code from Listing 9-3 so that the output starts at the number 3 and displays multiples of 3 all the way up to 100.

Exercise 9-7: Create a program that counts backward from 25 to 0.

Looping letters

Listing 9-4 shows another way to "count" using a `for` loop.

Listing 9-4: Counting by Letter

```
#include <stdio.h>

int main()
{
    char alphabet;

    for(alphabet='A';alphabet<='Z';alphabet=alphabet+1)
    {
        printf("%c",alphabet);
    }
    putchar('\n');
    return(0);
}
```

Before you type the source code from Listing 9-4, can you guess what the output might be? Does it make sense to you?

Exercise 9-8: Use the source code from Listing 9-4 to create a new project. Build and run.

Exercise 9-9: Modify the `printf()` function in Line 9 so that the `%d` placeholder is used instead of `%c`.

Computers see characters as numbers. Only when numbers are displayed and they fall in the ASCII code range for characters do characters appear. (See Appendix A for the list of ASCII character codes.)

Exercise 9-10: Using Listing 9-4 as your inspiration, write a `for` loop that "counts" backward from *z* (lowercase Z) to *a* (lowercase A).

Nesting for loops

One thing you can stick inside a `for` loop is another `for` loop. It may seem crazy to loop within a loop, but it's a common practice. The official jargon is *nested loop*. Listing 9-5 shows an example.

Listing 9-5: A Nested Loop

```
#include <stdio.h>

int main()
{
    int alpha,code;

    for(alpha='A';alpha<='G';alpha=alpha+1)
    {
        for(code=1;code<=7;code=code+1)
        {
            printf("%c%d\t",alpha,code);
        }
        putchar(,\n');      /* end a line of text */
    }
    return(0);
}
```

Don't let all the indents intimidate you; they make the code more readable. Indents also help show which statements belong to which `for` loop because they line up at the same tab stop.

Line 7 in Listing 9-5 begins the first, outer `for` loop. It counts from letters A to G. It also contains the second, inner `for` loop and a `putchar()` function on Line 13. That function helps organize the output into rows by spitting out a newline after each row is displayed.

The `printf()` function in Line 11 displays the program's output, specifying the outer loop value, `alpha`, and the inner loop value, `code`. The `\t` escape sequence separates the output.

Exercise 9-11: Type the source code from Listing 9-5 into your editor. Build and run.

Here's the output I see on my computer:

```
A1  A2  A3  A4  A5  A6  A7
B1  B2  B3  B4  B5  B6  B7
C1  C2  C3  C4  C5  C6  C7
D1  D2  D3  D4  D5  D6  D7
E1  E2  E3  E4  E5  E6  E7
F1  F2  F3  F4  F5  F6  F7
G1  G2  G3  G4  G5  G6  G7
```

A triple nested loop contains three `for` statements, which continues the cascade shown in Listing 9-5. As long as you can match up the curly brackets with each `for` statement (and that's easy, thanks to modern text editors), it's something you can accomplish quite readily.

Exercise 9-12: Write a three-letter acronym-generating program. The program's output lists all three-letter combinations from AAA through ZZZ, spewed out each on a line by itself.

I wrote a program similar to the solution to Exercise 9-12 as one of my first programming projects. The computers in those days were so slow that the output took about ten seconds to run. On today's computers, the output is nearly instantaneous.

The Joy of the while Loop

Another popular looping keyword in C is `while`. It has a companion, `do`, so programmers refer to this type of loop as either `while` or `do-while`. The C language is missing the `do-whacka-do` type of loop.

Structuring a while loop

The C language `while` loop is a lot easier to look at than a `for` loop, but it involves more careful setup and preparation. Basically, it goes like this:

```
while(condition)
{
    statement(s);
}
```

The *`condition`* is a true/false comparison, just like you'd find in an `if` statement. The *`condition`* is checked every time the loop repeats. As long as it's true, the loop spins and the statement (or statements) between the curly brackets continues to execute.

Because the evaluation (condition) happens at the start of the loop, the loop must be initialized before the `while` statement, as shown in Listing 9-6.

So how does a `while` loop end? The termination happens within the loop's statements. Usually, one of the statements affects the evaluation, causing it to turn false.

After the `while` loop is done, program execution continues with the next statement after the final curly bracket.

A `while` loop can also forgo the curly brackets when it has only one statement:

```
while(condition)
    statement;
```

Listing 9-6: The while Version of Listing 9-1

```
#include <stdio.h>

int main()
{
    int x;

    x=0;
    while(x<10)
    {
        puts("Sore shoulder surgery");
        x=x+1;
    }
    return(0);
}
```

The `while` loop demonstrated in Listing 9-6 has three parts:

- The initialization takes place on Line 7, where variable *x* is set equal to `0`.
- The loop's exit condition is contained within the `while` statement's parentheses, as shown in Line 8.
- The item that iterates the loop is found on Line 11, where variable *x* is increased in value. Or, as programmers would say, "Variable *x* is *incremented*."

Exercise 9-13: Create a new project, ex0913, using the source code from Listing 9-6. Build and run.

Exercise 9-14: Change Line 7 in the source code so that variable *x* is assigned the value 13. Build and run. Can you explain the output?

Exercise 9-15: Write a program that uses a `while` loop to display values from –5 through 5, using an increment of 0.5.

Using the do-while loop

The `do-while` loop can be described as an upside-down `while` loop. That's true, especially when you look at the thing's structure:

```
do
{
    statement(s);
} while (condition);
```

As with a `while` loop, the initialization must take place before entering the loop, and one of the loop's statements should affect the condition so that the loop exits. The `while` statement, however, appears after the last curly bracket. The `do` statement begins the structure.

Because of its inverse structure, the major difference between a `while` loop and a `do-while` loop is that the `do-while` loop is always executed at least one time. So you can best employ this type of loop when you need to ensure that the statements spin once. Likewise, avoid `do-while` when you don't want the statements to iterate unless the condition is true. (See Listing 9-7.)

Listing 9-7: A Fibonacci Sequence

```
#include <stdio.h>

int main()
{
    int fibo,nacci;

    fibo=0;
    nacci=1;

    do
    {
        printf("%d ",fibo);
        fibo=fibo+nacci;
        printf("%d ",nacci);
        nacci=nacci+fibo;
    } while( nacci < 300 );

    putchar('\n');
    return(0);
}
```

Exercise 9-16: Type the source code from Listing 9-7 into a new project, ex0916. Mind your typing! The final `while` statement (refer to Line 16) must end with a semicolon, or else the compiler gets all huffy on you.

Here's the output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

The loop begins at Lines 7 and 8, where the variables are initialized.

Lines 12 through 15 calculate the Fibonacci values. Two `printf()` functions display the values.

The loop ends on Line 16, where the `while` statement makes its evaluation. As long as variable *`nacci`* is less than `300`, the loop repeats. You can adjust this value higher to direct the program to output more Fibonacci numbers.

On Line 18, the `putchar()` statement cleans up the output by adding a newline character.

Exercise 9-17: Repeat Exercise 9-14 as a `do-while` loop.

Loopy Stuff

I could go on and on about loops all day, repeating myself endlessly! Before moving on, however, I'd like to go over a few looping tips and pratfalls. These things you should know before you get your official *For Dummies* Looping Programmer certificate.

Looping endlessly

Beware the endless loop!

When a program enters an endless loop, it either spews output over and over without end or it sits there tight and does nothing. Well, it's doing what you ordered it to do, which is to sit and spin forever. Sometimes, this setup is done on purpose, but mostly it happens because of programmer error. And with the way loops are set up in C, it's easy to unintentionally loop *ad infinitum*.

Listing 9-8 illustrates a common endless loop, which is a programming error, not a syntax error.

Listing 9-8: A Common Way to Make an Endless Loop

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0;x=10;x=x+1)
    {
        puts("What are you lookin' at?");
    }
    return(0);
}
```

The problem with the code in Listing 9-8 is that the `for` statement's exit condition is always true: `x=10`. Read it again if you didn't catch it the first time, or just do Exercise 9-18.

Exercise 9-18: Type the source code for Listing 9-8. Save, build, and run.

The compiler may warn you about the constant TRUE condition in the `for` statement. Code::Blocks should do that, and any other compiler would, if you ratcheted up its error-checking. Otherwise, the program compiles and runs — infinitely.

- To break out of an endless loop, press Ctrl+C on the keyboard. This trick works only for console programs, and it may not always work. If it doesn't, you need to kill the process run amok, which is something I don't have time to explain in this book.
- Endless loops are also referred to as *infinite loops*.

Looping endlessly but on purpose

Occasionally, a program needs an endless loop. For example, a microcontroller may load a program that runs as long as the device is on. When you set up such a loop on purpose in C, one of two statements is used:

```
for(;;)
```

I read this statement as "for ever." With no items in the parentheses, but still with the required two semicolons, the `for` loop repeats eternally — even after the cows come home. Here's the `while` loop equivalent:

```
while(1)
```

The value in the parentheses doesn't necessarily need to be `1`; any True or non-zero value works. When the loop is endless on purpose, however, most programmers set the value to `1` simply to self-document that they know what's up.

You can see an example of an endless loop on purpose in the next section.

Breaking out of a loop

Any loop can be terminated instantly — including endless loops — by using a `break` statement within the loop's repeating group of statements. When `break` is encountered, looping stops and program execution picks up with the next statement after the loop's final curly bracket. Listing 9-9 demonstrates the process.

Listing 9-9: Get Me Outta Here!

```
#include <stdio.h>

int main()
{
    int count;

    count = 0;
    while(1)
    {
        printf("%d, ",count);
        count = count+1;
        if( count > 50)
            break;
    }
    putchar('\n');
    return(0);
}
```

The `while` loop at Line 8 is configured to go on forever, but the `if` test at Line 12 can stop it: When the value of `count` is greater than 50, the `break` statement (refer to Line 13) is executed and the loop halts.

Exercise 9-19: Build and run a new project using the source code from Listing 9-9.

Exercise 9-20: Rewrite the source code from Listing 9-9 so that an endless `for` loop is used instead of an endless `while` loop.

You don't need to construct an endless loop to use the `break` statement. You can break out of any loop. When you do, execution continues with the first statement after the loop's final curly bracket.

Screwing up a loop

I know of two common ways to mess up a loop. These trouble spots crop up for beginners and pros alike. The only way to avoid these spots is to keep a keen eye so that you can spot 'em quick.

The first goof-up is specifying a condition that can never be met; for example:

```
for(x=1;x==10;x=x+1)
```

In the preceding line, the exit condition is false before the loop spins once, so the loop is never executed. This error is almost as insidious as using an assignment operator (a single equal sign) instead of the "is equal to" operator (as just shown).

Another common mistake is misplacing the semicolon, as in

```
for(x=1;x<14;x=x+1);
{
    puts("Sore shoulder surgery");
}
```

Because the first line, the `for` statement, ends in a semicolon, the compiler believes that the line is the entire loop. The empty code repeats 13 times, which is what the `for` statement dictates. The `puts()` statement is then executed once.

Those rogue semicolons can be frustrating!

The problem is worse with `while` loops because the `do-while` structure requires a semicolon after the final `while` statement. In fact, forgetting that particular semicolon is also a source of woe. For a traditional `while` loop, you don't do this:

```
while(x<14);
{
    puts("Sore shoulder surgery");
}
```

The severely stupid thing about these semicolon errors is that the compiler doesn't catch them. It believes that your intent is to have a loop without statements. Such a thing is possible, as shown in Listing 9-10.

Avoid goto hell

The third looping statement is the most despised and the lowest-of-the-low C language keywords. It's `goto`, which is pronounced "go to," not "gotto." It directs program execution to another line in the source code, a line tagged by label. Here's an example:

```
here:
    puts("This is a type of
   loop");
goto here;
```

As this chunk of code executes (from the top down), the `here` label is ignored. The `puts()` function comes next, and, finally, `goto` redirects program flow back up to the `here` label. Everything repeats. Everything works. But it's just darn ugly.

Most clever programmers can craft their code in ways that don't require `goto`. The end result is something more readable — and that's the key. Code that contains lots of `goto` statements can be difficult to follow, leading experienced programmers to describe it as *spaghetti code.* The `goto` statement encourages sloppy habits.

The only time `goto` is truly necessary is when busting out of a nested loop. Even in that situation, an example would be contrived. So it's probably safe to say that you'll run your entire programming career and, hopefully, never have to deal with a `goto` statement in C.

Listing 9-10: A for Loop with No Body

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0;x<10;x=x+1,printf("%d\n",x))
        ;
    return(0);
}
```

In the example shown in Listing 9-10, the semicolon is placed on the line after the `for` statement (refer to Line 8 in Listing 9-10). That shows deliberate intent.

You can see that two items are placed in the `for` statement's parentheses, both separated by a comma. That's perfectly legal, and it works, though it's not quite readable.

Exercise 9-21: Type the source code from Listing 9-10 into your editor. Build and run.

Though you can load up items in a `for` statement's parentheses, it's rare and definitely not recommended, for readability's sake.